

Department I - C Plus Plus

Modern and Lucid C++ for Professional Programmers

Week 5 – Classes and Operators

Thomas Corbat / Felix Morgner
Rapperswil, 16/19.10.2023
HS2023



- You can implement your own data types
- You know the elements a class consists of
- You know how to overload operators for classes
- You know the correct way to read and print objects

- **Recap Week 4**

- **Classes**

- Declaration / Implementation

- Access Specifiers

- Constructors

- Inheritance

- **Operators**

- Members

- Free Operators

Recap Week 4



- What would be the correct signature choice for printDocument, if the type of document

- ... is copyable
- ... potentially huge
- ... not modified in printDocument

```
auto printDocument(<Type> document) -> void {  
    for (auto const & line : document.content()) {  
        printLine(line);  
    }  
}
```

- Possibilities

- auto printDocument(Document document) -> void
- auto printDocument(Document const document) -> void
- auto printDocument(Document & document) -> void
- auto printDocument(Document const & document) -> void

- What is wrong with the following test case, that should check whether an `std::out_of_range` exception is thrown upon accessing an empty vector?

```
TEST(testForExceptionTryCatch) {  
    std::vector<int> empty_vector{};  
    try {  
        empty_vector.at(0);  
    } catch (std::out_of_range const & e) {  
    }  
}
```

Classes



Goals:

- You know how to define a class in C++
- You know the elements a class consists of
- You can implement your own data types

- **Does one thing well and is named after that**
 - High Cohesion
- **Consists of member functions with only a few lines**
 - Avoid deeply nested control structures
- **Has a class invariant**
 - Provides a guarantee about its state (values of the member variables)
 - Constructors establish that invariant
- **Is easy to use without complicated protocol sequence requirements**

GoodClassName.hpp

```
class <GoodClassName> {  
  
    <member variables>  
  
    <constructors>  
  
    <member functions>  
};
```


- A class defines a new type
- A class is usually defined in a header file
- At the end of a class definition a semicolon is required

```
#ifndef DATE_HPP_                                     Date.hpp
#define DATE_HPP_

class Date {
    int year, month, day;
public:
    Date(int year, int month, int day);

    static auto isLeapYear(int year) -> bool;

private:
    auto isValidDate() const -> bool;
};

#endif
```

- **Include guard ensures that the content of a header file is only included once**
 - Eliminates cyclic dependencies of `#include` directives
- **Prevents violation of the one definition rule**
- **Directives**
 - `#ifndef <name>`
 - `#define <name>`
 - `#endif`

```
#ifndef DATE_HPP_                                     Date.hpp
#define DATE_HPP_

class Date {
    int year, month, day;
public:
    Date(int year, int month, int day);

    static auto isLeapYear(int year) -> bool;

private:
    auto isValidDate() const -> bool;
};

#endif
```

- **Keywords for defining a class**

- class
- struct

- **Default visibility for members of the class are**

- private for class
- public for struct

```
struct <name> {  
    ...  
};
```

```
#ifndef DATE_HPP_                                     Date.hpp  
#define DATE_HPP_  
  
class Date {  
    int year, month, day;  
public:  
    Date(int year, int month, int day);  
  
    static auto isLeapYear(int year) -> bool;  
  
private:  
    auto isValidDate() const -> bool;  
};  
  
#endif
```

- **Access specifiers (followed by a colon :)**
 - **private:**
visible only inside the class (and friends); for hidden data members
 - **protected:**
also visible in subclasses
 - **public:**
visible from everywhere; for the interface of the class
- **All subsequent members have this visibility**
- **Each visibility can reoccur multiple times**

```
#ifndef DATE_HPP_                                     Date.hpp
#define DATE_HPP_

class Date {
    int year, month, day;
public:
    Date(int year, int month, int day);

    static auto isLeapYear(int year) -> bool;

private:
    auto isValidDate() const -> bool;
};

#endif
```

- Have a type and a name

`<type> <name>;`

- The content/state of an object that represents the value of the type
- Don't make member variables **const** as it prevents copy assignment
- Don't add members to communicate between member function calls
 - Hard to test
 - Such usage protocols are a burden for the user of the class
 - Better: Use parameters

```
#ifndef DATE_HPP_                                     Date.hpp
#define DATE_HPP_

class Date {
    int year, month, day;
public:
    Date(int year, int month, int day);

    static auto isLeapYear(int year) -> bool;

private:
    auto isValidDate() const -> bool;
};

#endif
```

- **Function with name of the class**

- Special member function

- **No return type**

```
<class name>(){} 
```

- **Initializer list for member initialization**

```
<class name>(<parameters>)  
: <initializer-list>  
{}
```

```
#ifndef DATE_HPP_                                     Date.hpp  
#define DATE_HPP_  
  
class Date {  
    int year, month, day;  
public:  
    Date(int year, int month, int day);  
  
    static auto isLeapYear(int year) -> bool  
    { /*...*/ }  
  
private:  
    auto isValidDate() const -> bool { /*...*/ }  
};  
  
#endif
```

● Default Constructor

```
Date d{};
```

- No parameters
- Implicitly available if there are no other declared constructors
- Has to initialize member variables with default values

● Copy Constructor

```
Date d2{d};
```

- Has one <own-type> const & parameter
- Implicitly available (unless there is a move constructor or assignment operator)
- Copies all member variables
- Usually, you don't need to implement it explicitly

```
class Date {  
public:  
    Date(int year, int month, int day);  
    //Default-Constructor  
    Date();  
    //Copy-Constructor  
    Date(Date const &);  
    //Move-Constructor  
    Date(Date &&);  
    //Destructor  
    ~Date();  
};
```



- **Move Constructor**

```
Date d2{std::move(d)};
```

- Has one <own-type> && parameter
 - Implicitly available (unless there is a copy constructor or assignment operator)
 - Moves all members
 - Usually, you don't need to implement it explicitly
- **Will be covered in C++ Advanced**

```
class Date {  
public:  
    Date(int year, int month, int day);  
    //Default-Constructor  
    Date();  
    //Copy-Constructor  
    Date(Date const &);  
    //Move-Constructor  
    Date(Date &&);  
    //Destructor  
    ~Date();  
};
```


- **Type-conversion Constructor**
- **Constructors that are callable with a single argument of a different type are called type-conversion constructors**
 - Constructors with a single parameter
 - Constructors with multiple parameter, which have default arguments for all parameters after the first
- **Type-conversion constructors should be declared explicit**
 - Avoids unexpected implicit conversion

```
explicit <ctor-name>(<OtherType>);
```

```
#ifndef DATE_HPP_
#define DATE_HPP_

class Date {
    int year, month, day;
    //...
public:
    explicit Date(int year,
                  int month = 1,
                  int day = 1);
};

#endif
```

Date.hpp

```
#include "Date.hpp"

auto printRemainingDays() -> void {
    Date today{2022, 10, 18};
    std::cout << today.daysUntil(Date{2023});
}
```

Calendar.cpp

- **Initializer List Constructor**

```
Container box{item1, item2, item3};
```

- Has one `std::initializer_list` parameter
- Does not need to be **explicit**, implicit conversion is usually desired

```
struct Container {  
    Container() = default;  
    Container(std::initializer_list<Element> elements);  
private:  
    std::vector<Element> elements{};  
};
```

- **Initializer List constructors are preferred if a variable is initialized with {}**

```
std::vector v(5, 10)
```

```
std::vector v{5, 10}
```

- **Named like the default constructor but with a leading ~**

```
~Date();
```

- **Must release all resources**
- **Implicitly available**
 - If you program properly, you will hardly ever need to implement it yourself!
- **Must not throw an exception!**
- **Called automatically at the end of the block for local instances**

```
class Date {  
public:  
    Date(int year, int month, int day);  
    //Default-Constructor  
    Date();  
    //Copy-Constructor  
    Date(Date const &);  
    //Move-Constructor  
    Date(Date &&);  
    //Typeconversion-Constructor  
    explicit Date(std::string const &);  
    //Destructor  
    ~Date();  
};
```

- Base classes are specified after the name

```
class <name> : <base1>, ..., <baseN>
```

- Multiple inheritance is possible
- Inheritance can specify a visibility
 - public, protected, private
 - Limits the **maximum** visibility of the inherited members
 - If no visibility is specified, the default of the inheriting class is used (class->private and struct->public)
- Details about Diamonds and Virtual inheritance later

```
class Base {  
    private:  
        int onlyInBase;  
    protected:  
        int baseAndInSubclasses;  
    public:  
        int everyoneCanFiddleWithMe  
};
```

```
class Sub : public Base {  
    //Can see baseAndInSubclasses and  
    //everyoneCanFiddleWithMe  
};
```

```
#ifndef DATE_HPP_
#define DATE_HPP_

class Date {
    int year, month, day;
public:
    Date(int year, int month, int day);

    static auto isLeapYear(int year) -> bool;

private:
    auto isValidDate() const -> bool;
};

#endif
```

Date.hpp

```
#include "Date.hpp"

Date::Date(int year, int month, int day)
    : year{year}, month{month}, day{day} {
    /*...*/
}

auto Date::isLeapYear(int year) -> bool {
    /*...*/
}

auto Date::isValidDate() const -> bool {
    /*...*/
}
```

Date.cpp

```
#include "Date.hpp"                                     Dating.cpp

auto dating() -> void {
    Date today{2016, 10, 19};

    auto thursday{today.tomorrow()};

    Date::isLeapYear(2016);

    //what now?
    Date invalidDate{2016, 13, 1};
}
```

```
#ifndef DATE_HPP_                                       Date.hpp
#define DATE_HPP_

class Date {
    int year, month, day;
public:
    Date(int year, int month, int day);

    auto tomorrow() const -> Date;

    static auto isLeapYear(int year) -> bool;

private:
    auto isValidDate() const -> bool;
};

#endif
```

```
struct Recipe {  
    Recipe(std::initializer_list<Step> steps);  
    auto cook() const -> Meal;  
private:  
    std::vector<Step> steps{};  
};
```

Correct

The class declaration has a semicolon at the end and a Recipe is constructible with a list of (cooking) steps. The visibilities are sensible.

```
struct Vehicle {  
    Location location{};  
};  
class Car : Vehicle {  
public:  
    auto drive(Destination destination) -> Route;  
};  
auto printLocation(Car& car) -> void {  
    std::cout << car.location;  
}
```

Inncorrect

While Vehicle and Car are syntactically correct, Car inherits privately from Vehicle (due to the class keyword). Therefore, the members of Vehicle cannot be accessed from outside the Car class.

- **Establish Invariant**

- Properties for a value of the type that are always true
- E.g. a Date instance always represents a valid date
- All (public) member functions assume and keep it intact

- **Initialize all members**

- Constructors only create a valid instance (Otherwise throw an exception!)
- Use initializer lists
- Use default values if possible/necessary

Date.cpp

```
#include "Date.hpp"

Date::Date(int year, int month, int day)
    : year{year}, month{month}, day{day} {
    if (!isValidDate()) {
        throw std::out_of_range{"invalid date"};
    }
}

Date::Date() : Date{1980, 1, 1} {
}

Date::Date(Date const & other)
    : Date{other.year, other.month, other.day} {
}
```


- As we have specified a default value, this should be the value created by the default constructor

- Constructor without parameters


- Remark regarding initialization:

```
Date nice_d{};  
Date ugly_d;
```

- Both create a Date and call the default constructor in this case
- The second (without {}) does not work with all types. It might contain uninitialized variables
- Good practice: Initialize all variables with {}!

```
#ifndef DATE_HPP_                                     Date.hpp  
#define DATE_HPP_  
  
class Date {  
    //...  
    Date();  
};  
  
#endif
```

```
#include "Date.hpp"                                   Date.cpp  
  
Date::Date()  
    : year{9999}, month{12}, day{31} {  
}
```

A blue arrow points from the `#include "Date.hpp"` line in the Date.cpp code block to the `#ifndef DATE_HPP_` line in the Date.hpp code block, indicating the inclusion relationship.

- **Member variables can have a default value assigned**

- NSDMI – Non-Static Data Member Initializers

```
class <classname> {
    <type> <membername>{<default-value>};
};
```

- **Such values are used if the member is not present in the initializer list of the constructor**
 - Initializer list still overrides those values
- **Useful if multiple constructors initialize data similarly**
 - Avoids duplication

```
#ifndef DATE_HPP_                                     Date.hpp
#define DATE_HPP_

class Date {
    int year{9999}, month{12}, day{31};
    //...
    Date();
    Date(int year, int month, int day);
};

#endif
```

```
#include "Date.h"                                     Date.cpp

Date::Date() {
}

Date::Date(int year, int month, int day)
    : year{year}, month{month}, day{day} {
    /*...*/
}
```

- **Some special member functions are implicitly available in certain cases**
 - E.g. Default constructor is implicitly available if no other explicit constructor is declared
- **(Re-)implementing the default behavior of the default constructor can be avoided by defaulting it**

```
<ctor-name>() = default;
```

- Adds the corresponding constructor to the type with the same behavior as if it was implicitly available
- Possible for:
 - Default constructor and destructor
 - Copy/move constructor
 - Copy/move assignment operator

```
#ifndef DATE_HPP_                                     Date.hpp
#define DATE_HPP_

class Date {
    int year{9999}, month{12}, day{31};
    //...
    Date() = default;
    Date(int year, int month, int day);
};

#endif
```

```
#include "Date.hpp"                                   Date.cpp

Date::Date(){
}

Date::Date(int year, int month, int day)
    : year{year}, month{month}, day{day} {
    /*...*/
}
```

- **Some special member functions are implicitly available in certain cases**
 - E.g. Default constructor is implicitly available if no other explicit constructor is declared
 - **Those implicit constructor are not always wanted**
 - Make them explicitly private
 - Or delete them
- `<ctor-name>() = delete;`
- Possible for:
 - Default constructor and destructor
 - Copy/move constructor
 - Copy/move assignment operator

```
Banknote.hpp
#ifndef BANKNOTE_HPP_
#define BANKNOTE_HPP_

class Banknote {
    int value;
    //...
    Banknote(Banknote & const) = delete;
};

#endif
```

```
Forger.cpp
#include "Banknote.hpp"

auto forge(Banknote const & note) -> Banknote {
    Banknote copy{note}; ///!
    return copy;
}
```

- Similar to Java constructors can call other constructors

C++

```
Ctor(Parameters)  
  : Ctor(Arguments) {  
}
```

Java

```
Ctor(Parameters) {  
    this(Arguments);  
}
```

- Constructor call has to be in the member initializer list
- Similar are calls to constructors of base classes

C++


```
Ctor(Parameters)  
  : Base(Arguments) {  
}
```

Java

```
Ctor(Parameters) {  
    super(Arguments);  
}
```

```
#ifndef DATE_HPP_                                     Date.hpp  
#define DATE_HPP_  
  
class Date {  
    //...  
    Date(int year, Month month, int day);  
    Date(int year, int month, int day);  
};  
  
#endif
```

```
#include "Date.hpp"                                   Date.cpp  
  
Date::Date(int year, int month, int day)  
    : Date{year, Month(month), day} {}
```



- **Don't violate invariant**
 - Leave object in valid state
- **Implicit `this` object**
 - Is a pointer
 - Member access with arrow: `->`
- **Declare `const` if possible!**
- **Must not modify members if `const`**
 - Refers to `this` object
 - Can only call `const` members
- **Otherwise access to all other members**

```
#include "Date.hpp"                                     Date.cpp

auto Date::isValidDate() const -> bool {
    if (day <= 0) {
        return false;
    }
    switch (month) {
        case 1: case 3: case 5: case 7:
        case 8: case 10: case 12:
            return day <= 31;
        case 4: case 6: case 9: case 11:
            return this->day <= 30;
        case 2:
            return day <= (isLeapYear(year) ? 29:28);
        default:
            return false;
    }
}
```

```
struct Counter {  
    auto increase(unsigned step) -> void {  
        auto before = current();  
        value = before + step;  
    }  
    auto current() const -> unsigned;  
private:  
    unsigned value;  
};
```

Correct

It is allowed to call const member functions from non-const member functions.

```
struct Document {  
    auto print(std::ostream & out) const -> void {  
        updatePrintDate();  
        out << content;  
    }  
private:  
    auto updatePrintDate() -> void;  
};
```

Inncorrect

It is not allowed to call a non-const member function from a const member function

- No **this** object
- Cannot be **const**
- No **static** keyword in implementation
- Call with `<classname>::<member>()`

```
Date::isLeapYear(2016);
```

```
class Date {                                     Date.hpp
    //...
    static auto isLeapYear(int year) -> bool;
};
```

```
#include "Date.hpp"                               Date.cpp

auto Date::isLeapYear(int year) -> bool {
    if (year % 400 == 0) {
        return true;
    }
    if (year % 100 == 0) {
        return false;
    }
    return year % 4 == 0;
}

//or the less readable version
auto Date::isLeapYear(int year) -> bool {
    return !(year % 4) &&
           ((year % 100) || !(year % 400));
}
```


- No **static** keyword in implementation
- **static const** member can be initialized directly
- Access outside class with name qualifier: `<classname>::<member>`

```
class Date {                                     Date.hpp
    static const Date myBirthday;
    static Date favoriteStudentsBirthday;
    static const Date today{2018, 10, 16};

    //...
};
```

```
#include "Date.hpp"                               Any.cpp
#include <iostream>

auto printBirthday() -> void {
    std::cout << Date::myBirthday;
    Date::favoriteStudentsBirthday = ...;
}
```

```
#include "Date.hpp"                               Date.cpp

Date const Date::myBirthday{1964, 12, 24};
Date Date::favoriteStudentsBirthday{1995, 5, 10};
```

```
struct Recipe {  
    Recipe(std::vector<Step> steps) = default;  
    auto cook() const -> Meal;  
private:  
    std::vector<Step> steps{};  
};
```

Incorrect

Of all constructors, only default, copy and move constructors can be declared = `default`.

```
struct Chair {  
    explicit Chair(unsigned legs = 4u);  
private:  
    unsigned legs;  
};  
Chair::Chair(unsigned legs)  
    : legs{legs} {  
    if (legs < 1u) {  
        throw std::invalid_argument{"..."};  
    }  
}
```

Correct

Constructors can have default arguments too. In the declaration, a constructor that can be called with a single argument should be explicit. It uses the member initializer list and throws an exception if the invariant cannot be established.

Operator Overloading



Goals:

- You know how to overload operators for classes
- You know the correct way to read and print objects
- You can deal with streams correctly in your classes

- Custom operators can be overloaded for user-defined types
- Declared like a function, with a special name

```
auto operator op(<parameters>) -> <returntype>
<returntype> operator op(<parameters>)
```

- Unary operators -> one parameter
 - Binary operators -> two parameters
- Implement operators reasonably!
 - Semantic should be natural
- "When in doubt, do as the `ints` do"
 - Scott Meyers – Effective C++

- Overloadable Operators (*op*):

+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]
<=>					

- Non-Overloadable Operators:

::	.*	.	?:
----	----	---	----

- Two dates are equal (==) if year, month and day are equal

```
auto congratulate(Date const& myBirthday, Date const& today) -> void {  
    if (myBirthday == today) {  
        out << "Happy Birthday!";  
    }  
}
```

- Dates can be ordered (<, >, <=, >=)

```
auto hasSemesterStarted(Date const& start, Date const& today) -> bool {  
    return start < today;  
}
```

- The three-way-comparison operator can be implemented to provide all relational (<, >, <=, >=) comparisons at once
- Also called «Spaceship Operator»

```
class Date {  
    int year, month, day;  
public:  
    auto operator<=>(Date const& right) const -> std::strong_ordering {  
        if (year != right.year) {  
            return year <=> right.year;  
        }  
        if (month != right.month) {  
            return month <=> right.month;  
        }  
        return day <=> right.day;  
    }  
    auto operator==(Date const& right) const -> bool {  
        return (*this <=> right) == std::strong_ordering::equal;  
    } //Also allows != comparison  
};
```

- The compiler can generate the three-way-comparison operator by defaulting it

- Three-way compares the members in declaration order
- This implicitly generates the equality operator as well

```
class Date {  
    int year, month, day;  
public:  
    auto operator<=>(Date const& right) const = default;  
};
```

- If you only want equality/inequality, that is possible as well

```
class Point {  
    int x, y;  
public:  
    auto operator==(Date const& right) const = default;  
};
```

- Two dates are equal (==) if year, month and day are equal

```
auto congratulate(Date const& myBirthday, Date const& today) -> void {  
    if (myBirthday == today) {  
        out << "Happy Birthday!";  
    }  
}
```

- Before C++20: Inequality (!=) had to be implemented along with equality

```
class Date {  
    int year, month, day;  
public:  
    auto operator==(Date const& right) const -> bool {  
        return year == right.year && month == right.month && day == right.day;  
    }  
    auto operator!=(Date const& right) const -> bool {  
        return !(*this == right);  
    }  
};
```


- Dates can be ordered (<, >, <=, >=)

```
auto hasSemesterStarted(Date const& start, Date const& today) -> bool {  
    return start < today;  
}
```

this

```
class Date {  
    int year, month, day;  
public:  
    auto operator<(Date const& right) const -> bool {  
        return year < right.year ||  
            (year == right.year && (month < right.month ||  
                (month == right.month && day == right.day)));  
    }  
    auto operator>(Date const& right) const -> bool { return right < *this; }  
    auto operator>=(Date const& right) const -> bool { return !(*this < right); }  
    auto operator<=(Date const& right) const -> bool { return !(right < *this); }  
};
```

```
class Date {
    int year, month, day;
public:
    auto operator<(Date const& right) const -> bool;
};

inline auto operator>(Date const& left, Date const& right) -> bool {
    return right < left;
}

inline auto operator>=(Date const& left, Date const& right) bool {
    return !(left < right);
}

inline auto operator<=(Date const& left, Date const& right) bool {
    return !(right < left);
}

inline auto operator==(Date const& left, Date const& right) bool {
    return !(left < right) && !(right < left);
}

inline auto operator!=(Date const& left, Date const& right) bool {
    return !(left == right);
}
```

Date.hpp

```
auto operator<=>(Date const& right) const -> std::strong_ordering;
```

- **Strong order**

- Values that are equivalent/equal are indistinguishable
- Either «a < b», «a == b» or «a > b» must be true
- For example ints or Dates

- **Values**

- std::strong_ordering::less for a < b
- std::strong_ordering::equivalent, std::strong_ordering::equal for a == b
- std::strong_ordering::greater for a > b

```
auto operator<=>(Date const& right) const -> std::weak_ordering;
```

- **Weak order**

- Values that are equivalent/equal may be distinguishable
- Either «a < b», «a == b» or «a > b» must be true
- For example Word when letter case is ignored

- **Values**

- `std::weak_ordering::less` for `a < b`
- `std::weak_ordering::equivalent` for `a == b`
- `std::weak_ordering::greater` for `a > b`

```
auto operator<=>(Date const& right) const -> std::partial_ordering;
```

● Partial order

- Values that are equivalent/equal may be distinguishable
- «a < b», «a == b» or «a > b» can all be false
- For example double as NaN always compares false even with itself! `std::isnan` is required

● Values

- `std::partial_ordering::less` for `a < b`
- `std::partial_ordering::equivalent` for `a == b`
- `std::partial_ordering::greater` for `a > b`
- `std::partial_ordering::unordered` for none of the above

- **Output operator must be a free function**

- `operator<<`
- Parameters: `std::ostream&` and `Date const&`
- Returns `std::ostream&` for chaining output

```
#include "Date.hpp"                                     Any.cpp
#include <iostream>

auto printBirthday() -> void {
    std::cout << Date::myBirthday;
}
```

```
#include <ostream>                                         Date.hpp

class Date {
    int year, month, day;
public:
    auto print(std::ostream& os) const -> void {
        os << year << "/" << month << "/" << day;
    }
};

inline auto operator<<(std::ostream& os, Date const& date) -> std::ostream& {
    return date.print(os);
}
```

- Input operator must be a free function

- `operator>>`
- Parameters: `std::istream&` and `Date&`
- Returns `std::istream&` for chaining input

```
#include "Date.hpp" Any.cpp
#include <iostream>

auto readDate() -> Date {
    Date date{};
    std::cin >> date;
    return date;
}
```

```
#include <istream> Date.hpp

class Date {
    int year, month, day;
public:
    Date(int year, int month, int day);
};

inline auto operator>>(std::istream& is, Date& date)
    -> std::istream& {
    int year{-1}, month{-1}, day{-1};
    char sep1, sep2;
    is >> year >> sep1 >> month >> sep2 >> day;
    try {
        date = Date{year, month, day};
        is.clear();
    } catch (std::out_of_range const& e) {
        is.setstate(std::ios::failbit);
    }
    return is;
}
```

```
struct SwissGrid {
    SwissGrid() = default;
    SwissGrid(double y, double x);
    auto read(std::istream& in) -> void {
        double y{}; in >> y;
        double x{}; in >> x;
        try {
            SwissGrid inputCoordinate{y, x};
            *this = inputCoordinate;
        } catch(std::invalid_argument const& e) {
            in.setstate(std::ios_base::failbit);
        }
    }
private:
    double y{600000.0};
    double x{200000.0};
};
inline auto operator>>(std::istream& in,
                       SwissGrid& coordinate) -> std::istream& {
    coordinate.read(in);
    return in;
}
```

Correct

The input (and output) operator must be implemented as free function. Since they usually won't have access to the private members of a type, a member function for reading/writing is required.

- **Separate the class declaration from the member function implementations properly into header and source files**
- **Initialize member variables with default values or in the constructor's initializer list**
- **Throw an exception from a constructor if it cannot establish the class invariant**
- **You need to implement input and output operators as free functions**
- **Provide sensible operations when implementing operators for your types**

- **Added range algorithm overloads**
- **Significant changes to the whole Operators chapter**